ODEs, PDEs, and Fourier Transforms!

PHYS 250 (Autumn 2018) – Lecture 11

David Miller

Department of Physics and the Enrico Fermi Institute
University of Chicago

November 8, 2018

Outline

Reminders from last time

We discussed the concepts that grew out of (or built upon) root finding – manipulation of Taylor series approximations – in order to expand our computational toolkit

Application of root finding and numerical differentiation

- **Newton's method:**
  - Pathologies associated with the naive implementation, as well as simple modifications that can mitigate issues (**backtracking**)
  - Multidimensional applications in matrix form and systems of equations
- **Ordinary differential equations:**
  - Obtained and discussed the Runge-Kutta algorithm(s)
  - Differentiated between **initial** and **boundary** value problems

Today we will take the next steps with ODEs and transition to PDEs!

Outline

Runge-Kutta family of algorithms

The aim of Runge-Kutta methods is to eliminate the need for repeated differentiation of the differential equations. Because no such differentiation is involved in the **first-order Taylor series** expression:

$$\vec{y}(x+h) = \vec{y}(x) + \vec{y}'(x)h = \vec{y}(x) + \vec{F}(x,\vec{y})h \tag{1}$$

This first-order version is referred to as **Euler's method (aka Euler's Rule)**.

Effectively, what this is doing is to start with the known initial value of the dependent variable, $y_0 \equiv y(x=0)$, and then use the derivative function $f(x,y)$ to find an approximate value for y at a small step $x=h$ forward in time; that is, $y(x=h) \equiv y_1$.

We know from our discussion of differentiation that the error in the forward-difference algorithm is $\mathcal{O}(h)$, and so this too is the error in Euler's rule.

How can we test this?

Code up the algorithm!

ExplicitEuler

```
    for i, x_i in enumerate(x[:-1]):

        h = x[i+1] - x_i
        y[i+1] = y[i] + h*func(x_i, y[i], args)

    return y
```

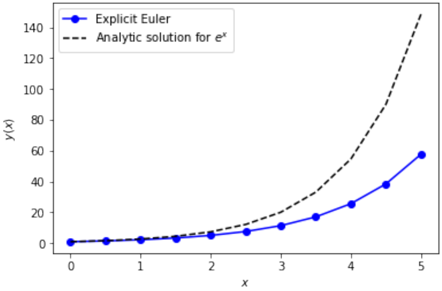
Code up the algorithm!

ExplicitEuler

```
    for i, x_i in enumerate(x[:-1]):

        h = x[i+1] - x_i
        y[i+1] = y[i] + h*func(x_i, y[i], args)

    return y
```



Improve the accuracy

We know that the accuracy depends on h so make that smaller?

ExplicitEuler

```
for N in [5, 10, 20, 40]:
    x = np.linspace(0., x_max, N+1) # Time steps
    y = ExplicitEuler(exp, y_0, x, solve_args)
    plt.plot(x, y, '-o', label='%d steps'%N)

plt.plot(x,np.exp(solve_args['a']*x),'k--',label='Known')
plt.xlabel(r'$x$'), plt.ylabel(r'$y$')
plt.legend(loc=2)
```

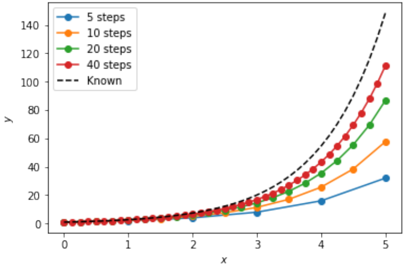
Improve the accuracy

We know that the accuracy depends on h so make that smaller?

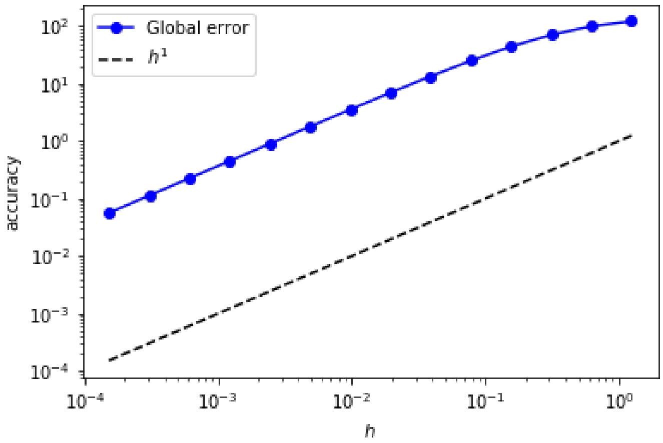
ExplicitEuler

```
for N in [5, 10, 20, 40]:
    x = np.linspace(0., x_max, N+1) # Time steps
    y = ExplicitEuler(exp, y_0, x, solve_args)
    plt.plot(x, y, '-o', label='%d steps'%N)

plt.plot(x,np.exp(solve_args['a']*x),'k--',label='Known')
plt.xlabel(r'$x$'), plt.ylabel(r'$y$')
plt.legend(loc=2)
```



Error of the method



Second order Runge-Kutta family of algorithms

Recall that the key insight was to expand $f(x,y)$ in a Taylor series about the **midpoint of the integration interval** and retain two terms:

$$f(x,y) \simeq f(x_{n+1/2}, y_{n+1/2}) + (x - x_{n+1/2})\frac{df}{dx}(x_{n+1/2}) + \mathcal{O}(h^2) \qquad (2)$$

As you recall from the **finite central difference** algorithm, only **odd powers** of h remain, and thus when used inside the integral above, the terms with $(x - x_{n+1/2})^{n\in\text{odd}}$ vanish. We are left with

$$y_{n+1} \simeq y_n + hf(x_{n+1/2}, y_{n+1/2}) + \mathcal{O}(h^2) \qquad (3)$$

How can we test this?

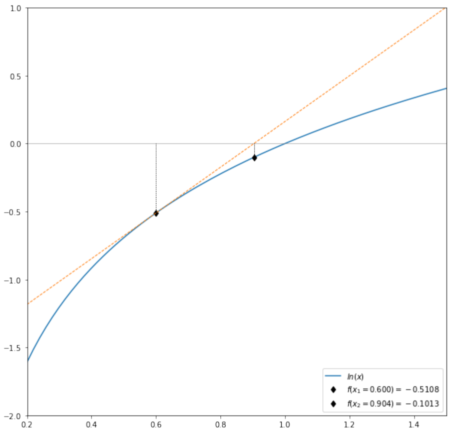
Precision of Newton's method

For any estimate x_i of the method, the error, E_i is the difference between the true root x and the estimate:

$$E_i = x - x_i \tag{4}$$

Merely by inspecting the design of Newton's method, you can see that the precision of the estimate for a subsequent iteration will be given by

$$\begin{aligned} E_{i+1} &= E_i + \frac{F(x)}{F'(x)} \quad (5) \\ &= -\frac{F''(x)}{2F'(x)} E_i^2 \quad (6) \end{aligned}$$

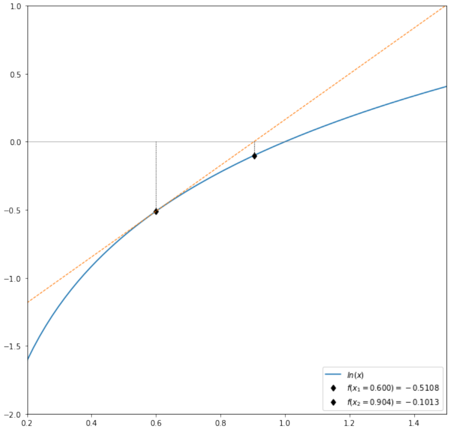


Convergence of Newton's method

Consequently, Newton's method **converges quadratically**

- the error is the square of the error in the previous step)
- the number of significant figures is roughly doubled in every iteration, provided that x_i is **sufficiently** close to the root.

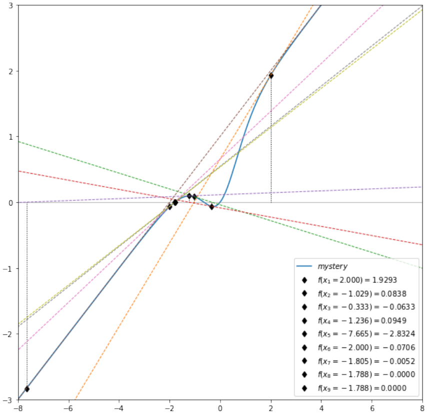
However, a critical assumption is that $F'(x) \neq 0$; for all $x \in I$, where I is the interval $[x - r, x + r]$ for some $r \geq |x - x_0|$ and x is the true root and x_0 was the starting point.



Pathologies and divergent scenarios

That is definitely not always the case. Let's look at a pathological example. Here is a fun mystery function that I cooked up (since you need to do something similar on your homework):

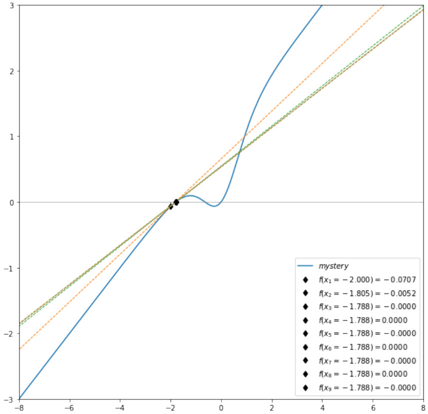
- $x_0 = 2.000$, **7 iterations**
- $x_0 = -2.000$, **2 iterations**
- $x_0 = 3.000$, **2 iterations**
- $x_0 = -1.214$, **4 iterations**
- Slight modification: $x_0 = -1.213$, **no convergence**
- Slight modification: $x_0 = 3.000$, **no convergence**



Pathologies and divergent scenarios

That is definitely not always the case. Let's look at a pathological example. Here is a fun mystery function that I cooked up (since you need to do something similar on your homework):

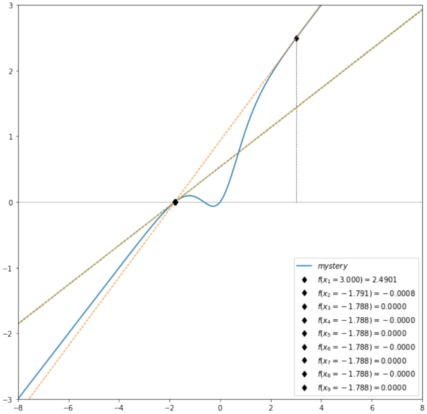
- $x_0 = 2.000$, **7 iterations**
- $x_0 = -2.000$, **2 iterations**
- $x_0 = 3.000$, **2 iterations**
- $x_0 = -1.214$, **4 iterations**
- Slight modification: $x_0 = -1.213$, **no convergence**
- Slight modification: $x_0 = 3.000$, **no convergence**



Pathologies and divergent scenarios

That is definitely not always the case. Let's look at a pathological example. Here is a fun mystery function that I cooked up (since you need to do something similar on your homework):

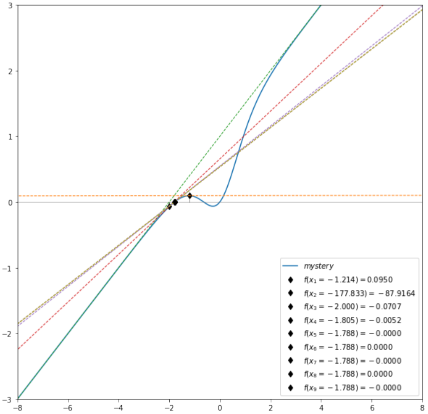
- $x_0 = 2.000$, **7 iterations**
- $x_0 = -2.000$, **2 iterations**
- $x_0 = 3.000$, **2 iterations**
- $x_0 = -1.214$, **4 iterations**
- Slight modification: $x_0 = -1.213$, **no convergence**
- Slight modification: $x_0 = 3.000$, **no convergence**



Pathologies and divergent scenarios

That is definitely not always the case. Let's look at a pathological example. Here is a fun mystery function that I cooked up (since you need to do something similar on your homework):

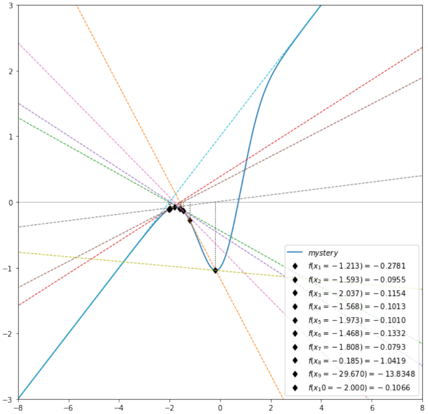
- $x_0 = 2.000$, **7 iterations**
- $x_0 = -2.000$, **2 iterations**
- $x_0 = 3.000$, **2 iterations**
- $x_0 = -1.214$, **4 iterations**
- Slight modification: $x_0 = -1.213$, **no convergence**
- Slight modification: $x_0 = 3.000$, **no convergence**



Pathologies and divergent scenarios

That is definitely not always the case. Let's look at a pathological example. Here is a fun mystery function that I cooked up (since you need to do something similar on your homework):

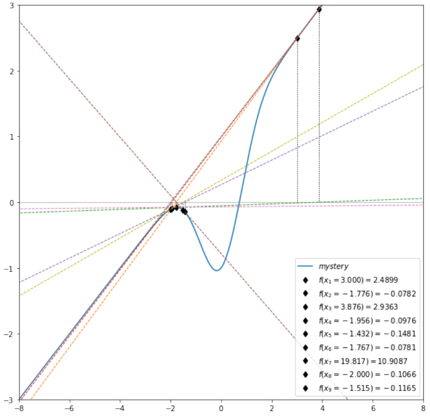
- $x_0 = 2.000$, **7 iterations**
- $x_0 = -2.000$, **2 iterations**
- $x_0 = 3.000$, **2 iterations**
- $x_0 = -1.214$, **4 iterations**
- Slight modification:
  $x_0 = -1.213$, **no convergence**
- Slight modification: $x_0 = 3.000$, **no convergence**



Pathologies and divergent scenarios

That is definitely not always the case. Let’s look at a pathological example. Here is a fun mystery function that I cooked up (since you need to do something similar on your homework):

- $x_0 = 2.000$, **7 iterations**
- $x_0 = -2.000$, **2 iterations**
- $x_0 = 3.000$, **2 iterations**
- $x_0 = -1.214$, **4 iterations**
- Slight modification: $x_0 = -1.213$, **no convergence**
- Slight modification: $x_0 = 3.000$, **no convergence**



Backtracking

In the last examples above we have a case where the search falls into the pathology of a situation where the initial guess was not **sufficiently close** to the root. an "infinite" loop without ever getting there.

A solution to this problem is called **backtracking**.

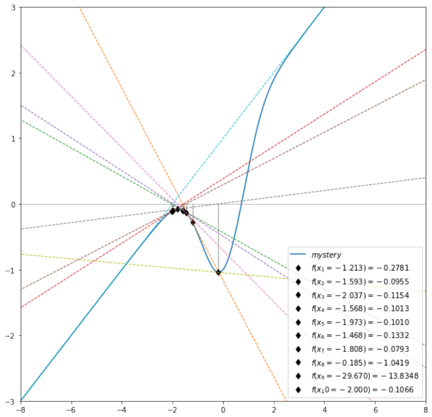
Backtracking

In cases where the new guess $x_0 + \Delta x$ leads to an increase in the magnitude of the function, $|f(x_0 + \Delta x)|^2 > |f(x_0)|^2$, you should backtrack somewhat and try a smaller guess, say, $x_0 + \Delta x/2$. If the magnitude of f still increases, then you just need to backtrack some more, say, by trying $x_0 + \Delta x/4$ as your next guess, and so forth.

Pathological case fixed with backtracking

Fixing the pathological example with backtracking:

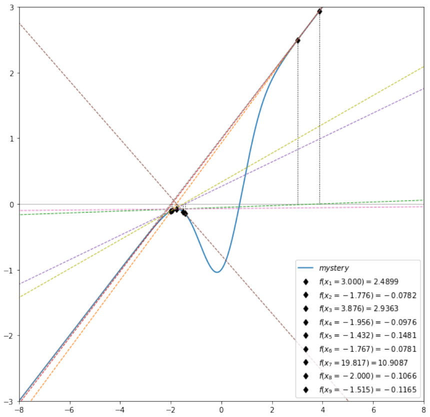
- $x_0 = -1.213$, **no convergence**
- $x_0 = 3.000$, **no convergence**
- $x_0 = 1.500$, **3 iterations**



Pathological case fixed with backtracking

Fixing the pathological example with backtracking:

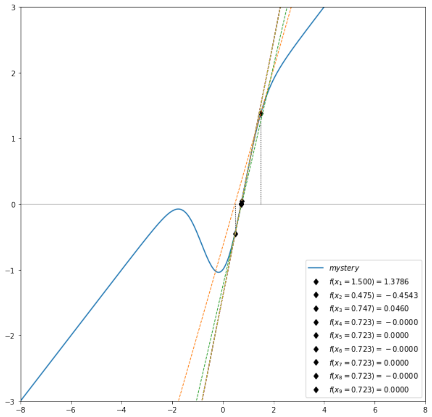
- $x_0 = -1.213$, **no convergence**
- $x_0 = 3.000$, **no convergence**
- $x_0 = 1.500$, **3 iterations**



Pathological case fixed with backtracking

Fixing the pathological example with backtracking:

- $x_0 = -1.213$, **no convergence**
- $x_0 = 3.000$, **no convergence**
- $x_0 = 1.500$, **3 iterations**



Multidimensional problems

Up to this point, we have confined our attention to solving the single equation $F(x) = 0$. Let us now consider the n-dimensional version of the same problem, namely

$$\vec{F}(\vec{x}) = 0 \tag{7}$$

where we allow for a vector of functions $\vec{F} = \{f_1(\vec{x}), f_2(\vec{x}), ..., f_n(\vec{x})\}$, and $\vec{x} = \{x_1, x_2, ..., x_n\}$.

The solution of n simultaneous, nonlinear equations is a much more formidable task than finding the root of a single equation. The trouble is the lack of a reliable method for bracketing the solution vector $\vec{x}$. Therefore, we cannot always provide the solution algorithm with a good starting value of x, unless such a value is suggested by the physics of the problem.

Newton's method is the workhorse here!

Outline

General form and structure of ODEs

$$\nabla^2 V = 0. \tag{8}$$

Initial vs. boundary value problems

The solution now requires the knowledge of n conditions to fully specify. If these conditions are specified at **the same value of x**, the problem is **an initial value problem**. Then the **initial conditions**, have the form:

$$\begin{aligned} y_0(a) &= \alpha_0 \\ y_1(a) &= \alpha_1 \\ &\vdots \\ y_{n-1}(a) &= \alpha_{n-1} \end{aligned}$$

On the other hand, if y_i are specified **at different values of x,** the problem is a **boundary value problem**.

$$\begin{aligned} y'' &= -y \\ y(0) &= 1 \\ y(\pi) &= 0 \end{aligned}$$

Recall the Taylor series approach generally

Writing the conditions for the initial value problem from the previous slides as

$$\vec{y}' = \vec{F}(x, \vec{y}), \qquad \vec{y}(a) = \vec{\alpha} \tag{9}$$

where

$$F(x, \vec{y}) = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ f(x, \vec{y}) \end{bmatrix}, \tag{10}$$

We can sort of "brute force" our way through it by repeatedly computing derivatives numerically using Newton's method. That's perfectly valid, but there are also better ways to go about it.

Avoiding repeated differentiation: Runge-Kutta

The aim of Runge-Kutta methods is to eliminate the need for repeated differentiation of the differential equations. Because no such differentiation is involved in the **first-order Taylor series** expression:

$$\vec{y}(x+h) = \vec{y}(x) + \vec{y}'(x)h = \vec{y}(x) + \vec{F}(x,\vec{y})h \tag{11}$$

This first-order version is referred to as **Euler's method (aka Euler's Rule)**.

Effectively, what this is doing is to start with the known initial value of the dependent variable, $y_0 \equiv y(x=0)$, and then use the derivative function $f(x,y)$ to find an approximate value for y at a small step $x=h$ forward in time; that is, $y(x=h) \equiv y_1$.

We know from our discussion of differentiation that the error in the forward-difference algorithm is $\mathcal{O}(h)$, and so this too is the error in Euler's rule.

Second-order Runge-Kutta algorithm

The Runge-Kutta algorithms for integrating a differential equation are based upon the formal (exact) integral of our differential equation:

$$\frac{dy}{dx} = f(x,y) \Rightarrow y(x) = \int f(x,y)dx \tag{12}$$

And therefore

$$y_{n+1} = y_n + \int_{x_n}^{x_{n+1}} f(x,y)dx \tag{13}$$

The key insight is to expand $f(x,y)$ in a Taylor series about the **midpoint of the integration interval** and retain two terms:

$$f(x,y) \simeq f(x_{n+1/2}, y_{n+1/2}) + (x - x_{n+1/2})\frac{df}{dx}(x_{n+1/2}) + \mathcal{O}(h^2) \tag{14}$$

As you recall from the **finite central difference** algorithm, only **odd powers** of h remain, and thus when used inside the integral above, the terms with $(x - x_{n+1/2})^{n \in \text{odd}}$ vanish. We are left with

$$y_{n+1} \simeq y_n + hf(x_{n+1/2}, y_{n+1/2}) + \mathcal{O}(h^3) \tag{15}$$

Difficulty with the second-order Runge-Kutta algorithm

The price for improved precision is having to evaluate the derivative function and y at the middle of the interval, $x = x_n + h/2$.

And there's the rub: we don't know the value of $y_{n+1/2}$ and cannot use this algorithm to determine it.

The way out of this issue is to use Euler's algorithm for $y_{n+1/2}$:

$$y(x + h/2) = y_n + \frac{1}{2}h\vec{y}' = y_n + \frac{1}{2}hf(x_n, y_n) \tag{16}$$

In this way, the known derivative function f is evaluated at the ends and the midpoint of the interval, but that only the (known) initial value of the dependent variable y is required. This makes the algorithm self-starting.

$$y_{n+1} \simeq y_n + k_2 \tag{17}$$

where

$$k_2 = h\vec{f}(x_n + \frac{h}{2}, \vec{y}_n + \frac{k_1}{2}), \qquad k_1 = h\vec{f}(x_n, y_n) \tag{18}$$

Precision

As discussed, the second order Runge-Kutta only achieves an $\mathcal{O}(h^3)$ precision. That precision is quickly insufficient for many applications.

The fourth-order Runge-Kutta method achieves an $\mathcal{O}(h^4)$ precision by approximating y as a Taylor series up to h^2 (a parabola) at the midpoint of the interval.

This approximation provides an excellent balance of power, precision, and programming simplicity. There are now four gradient terms to evaluate with four subroutine calls needed to provide a better approximation to $f(x, y)$ near the midpoint.

Higher order calculations: `rk4`

Without deriving anything, I simply write out the definition and then we'll elaborate on the implications

$$y_{n+1} \simeq y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{19}$$

where

$$\begin{aligned}
k_1 &= h\vec{f}(x_n, y_n) && (20)\\
k_2 &= h\vec{f}\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) && (21)\\
k_3 &= h\vec{f}\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) && (22)\\
k_4 &= h\vec{f}\left(x_n + h, y_n + k_3\right) && (23)
\end{aligned}$$

The benefit of the calculation of the additional terms is a $\mathcal{O}(h^4)$ precision on the final results.

Outline

The issue with boundary value problems

In an initial value problem we were able to start at the point where the initial values were given and march the solution forward as far as needed.

This technique does not work for boundary value problems, because there are not enough starting conditions available at either endpoint to produce a unique solution.

The simplest two-point boundary value problem is a second-order differential equation with one condition specified at $x = a$ and another one at $x = b$. Here is an example of such a problem:

$$y'' = f(x, y, y'), \qquad y(a) = \alpha, \qquad y(b) = \beta \tag{24}$$

The whole point is to attempt to turn these equations into the initial value problem such that

$$y'' = f(x, y, y'), \qquad y(a) = \alpha, \qquad y' = u \tag{25}$$

and the key to success is finding the correct value of u.

Root finding

Really, this is just a matter of **root finding**!! Which, of course, you now know very well how to do!

The solution of the initial value problem depends on u, the computed value of $y(b)$ is a function of u; that is,

$$y(b) = \theta(u) \tag{26}$$

And hence, u is a root of the expression:

$$r(u) = \theta(u) - \beta = 0 \tag{27}$$

where $r(u)$ is the boundary residual (difference between the computed and specified boundary value at $x = b$)

This is the essence of the shooting method, which we will discuss next time